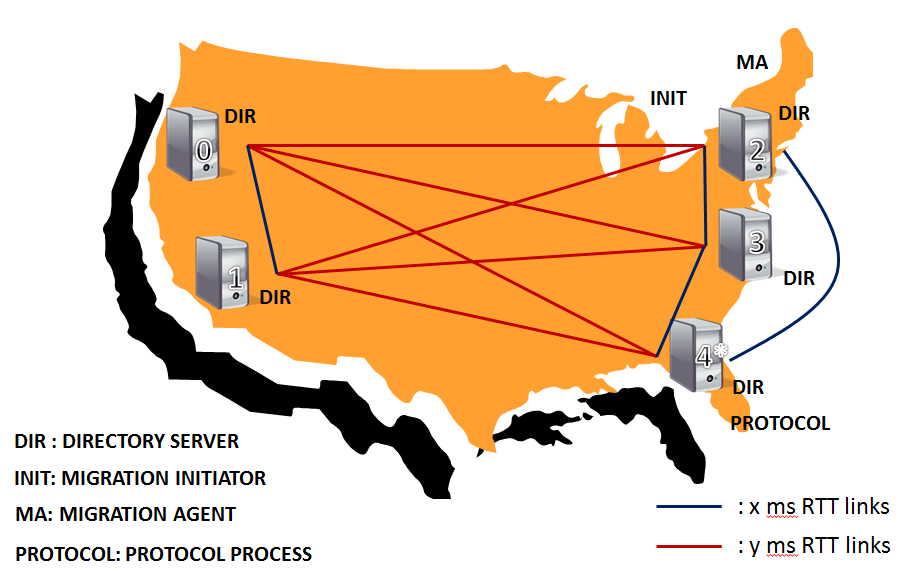
1. Introduction
2. Background
   1. Paxos

* An algorithm for implementing fault-tolerant distributed systems.
* The heart is a consensus algorithm – how do we get multiple processes that are each trying to assert/propose a value to agree upon and stick with a single value?
* The safety requirements:
  + Only a single value that has been proposed may be chosen
  + Processes learn about values if and only if they have been chosen
* The protocol does not specify any liveness/convergence requirements.
* There are 3 classes of “agents” that take part in the protocol:
  + Proposers – They propose values to be chosen
  + Acceptors – They choose to or not to accept proposed values
  + Learners – They learn the final, single proposed value that was accepted by the acceptors (not all, just a majority, see below)
  + There are no strict requirements on the mappings between the given processes and these roles
* A proposed value can be considered accepted once a majority of acceptors have accepted it.
* The cornerstone of the algorithm lies in determining how and which value must be accepted.
* From a bird’s eye perspective, the acceptors control the proposers and their proposed values – so the working of the algorithm is driven by acceptors forcing the proposers to propose acceptable values, whilst the design of the algorithm revolves around setting down rules for how to accept values.
* The design considerations for accepting values are as follows (revised as new requirements emerge:
  + 1. An acceptor must accept the first proposal it receives – we must begin somewhere
  + Only a single value must be accepted => we’ll turn this around and instead put the responsibility on the proposers and say – only that value may be proposed repeatedly
  + As a proposer, I can see what values have been accepted while proposing, but I cannot predict what values might be accepted in the future. To this end, I somehow seek to control the future acceptances by extracting promises from acceptors regarding the nature of the same
    - Proposals now have a proposal number. To avoid confusions, different proposals must have different numbers, a global ordering of sort – the implementation left open ended. A suggestion would be to just have proposers choose the numbers from non-overlapping sequences and store the last used number in stable storage.
    - Promise to me, the proposer that you, the acceptor will not accept a proposal with a number lower than mine
    - If you have already accepted a proposal, let me know.
  + Due to this extracted promise, we need to change acceptance rule 1 to: 1a. Acceptors can and must only accept proposals that do not violate promises it has made => accept proposals which have numbers > numbers of proposals to which promises have been made
  + 2. If a proposal with value ‘v’ is chosen, then every higher numbered proposal that is chosen by any acceptor has value ‘v’ – this follows from the requirement that only a single value be chosen in a round of Paxos.
  + This is where the implementation of the algorithm is driven backwards – to ensure that no proposal with a value other than ‘v’ with a proposal number higher than the highest accepted proposal number (with value ‘v’) is accepted, the acceptors force the proposer to only issue proposals with value ‘v’. Hence:   
    2a. If a proposal with value ‘v’ is chosen, then every higher-numbered proposal issued by any proposer has value ‘v’.
  + Now we relax constraint 2a by moving to a majority instead of every acceptor. Hence:   
    2b. For any proposal numbered ‘n’ with value ‘v’ issued, there exists a set ‘S’ consisting of a majority of acceptors such that either:
    - a) no acceptor in S has accepted any proposal numbered less than ‘n’, or
    - b) ‘v’ is the value of the highest numbered proposal among all proposals numbered less than ‘n’ accepted by acceptors in ‘S’
* Putting all this together, the algorithm for a single ‘round’ of Paxos sums up to such:   
  Phase 1.   
  (a) A proposer selects a globally exclusive proposal number ‘n’ and sends a prepare request to a majority of acceptors (it could be all acceptors in the implementation) – this is called a ‘prepare’ request.  
  (b) If an acceptor receives a ‘prepare’ request with number ‘n’ greater than any ‘prepare’ request to which it has already responded, it responds to the request with a promise not to accept any more proposals with number less than ‘n’, and the number ‘n’ and value ‘v’ of the highest number proposal it has accepted (if any).  
  Phase 2.  
  (a) If the proposer receives a response to its prepare request numbered ‘n’ from a majority of acceptors, then it sends an ‘accept’ request to each of those acceptors for a proposal numbered ‘n’ with either the value of the highest numbered proposal it received from the acceptors in response to its prepare request, or if no such value exists, then any value of its choosing.  
  (b) If an acceptor receives an accept request for a proposal numbered ‘n’ >= highest prepare request number it has responded to, then it accepts the proposal.
* A few things to note:
  + There is no direct correlation between Phases 1 and 2 in terms of a Phase 1 being sufficient for Phase 2. That is, a proposer ‘P1’ could elicit a response to its prepare request but it might end up racing with another proposer ‘P2’ in that acceptors could end up rejecting P1’s accept requests after accepting its prepare requests because P2 is racing P1 and keeps issuing prepare requests with numbers succeeding P1’s prepare requests.
  + A decision is implicitly reached when a majority of the acceptors accept the same value ‘v’ – because using induction and the property of there being at least one common acceptor in the intersection of 2 majorities of acceptors, we can show that the acceptors will force any future proposers into re-proposing the same accepted value.
  + There is no limit on the number of proposals that can be made – proposers can abandon proposals mid-flight and reissue proposals of higher numbers if they want.
  + There is no guarantee of convergence – the protocol is correct, but may never converge.
* To learn a chosen value, the learners must find out that a majority set of acceptors have accepted a single value. There are multiple ways to do this, the most straightforward of which would be to have every acceptor ack acceptances it makes to every learner.
* Optimizations:
  + The first obvious optimization would be some step to alleviate the non-convergence problem. We could have a “distinguished proposer”, a leader who would be the only one trying to issue proposals, circumventing the race problem.
  + Similarly, we could have a distinguished learner, or a set of them to reduce the number of acks that the acceptors would have to send out once they accept a value.

1. Design and Implementation
   1. Dexter and DTunes?
   2. The Migration Protocol  
      
   3. Logging Framework

1. Experimental Methodology and Results
   1. Aim
      * The high level aim of the experiments was to get a deeper understanding of the Paxos algorithm in an actual implementation.
      * This effort can be broadly classified into 2 steps:
        + Understanding the implementation and it’s departures from the algorithm in terms of optimizations and details generally left open to implementation.
        + Instrumenting the implementation to piece-wise analyze the different component times in the algorithm.
      * Coupling this with different delays simulated between replicas would give us an example of an instrumented Paxos system deployed in a WAN.
      * The Directory Protocol gives us a working state machine system built around the Paxos core to drive the experiments.
   2. Setup  
      The experimental is as shown below:  
        
      

* We used a 5 replica cluster.
* The cluster was split into 2 sub-clusters. Depending on the inter and intra cluster delays this allowed us to model different geographical setups.
* Each replica runs its own directory server
* The Migration Initiator is co-located with the Migration Agent on one of the replicas. We did not want to end up needing a lot of the machines to conduct these experiments. Also since all 3 processes are rather lightweight, there would be limited overhead/contention between them.
* The Protocol Process runs on every machine, but since for the purpose of these experiments we are not simulating any failures, all processes apart from the one co-located with the leader replica are of no consequence as they would be unable to take any action.
* Since the focus here is on latency and not throughput measurements, we set the parameters to have high polling frequencies, and low critical path latencies.
  + CrashModel – EpochSS
  + WindowSize – 2 (the migrations are serial for the purpose of these experiments)
  + MaxBatchDelay – 0 (do not batch, push proposals instantly on arrival)
* The implementation had been modified to force any Paxos client to always connect to the leader process (directly, no redirections) and the same replica (#4) is elected leader for all experiments. This forces uniformity between runs so we can set strong expectations for our experiments.
  1. Experiments
     + The experiments are each run for 10 key migrations – each key migration involves 5 Paxos stages, namely:
       - Migration Initiation
       - Directory Acks – 1 per agent (5 in our implementation)
       - Updating Timestamp after informing Migration Agent about object move
       - Migration Agent Acks – 1 per agent (1 in our implementation)
       - Updating migration record to complete
     + All graphs are plotted on the basis of above mentioned Paxos rounds.
     + We ran our experiments for 3 different network configurations. Discussed below are each configuration and associated expectations and results.
     + No DummyNet:
       - These set of experiments are running on the PRObE testbed’s Infiniband fabric directly.
       - The link delays between nodes i.e. x = y = ~0.1ms.
       - Expectations:
         1. The Paxos latencies as measured from the enqueued phase to decide phase measured at the leader (as that is the only replica which has to power to queue and propose) should be of the order of a few milliseconds. This is heavily dominated mainly by 2 factors:

Code execution time and threading overhead.

Queuing latencies between the multiple asynchronous parts of the application

* + - * 1. The client end to end latencies should be dominated by the service time of the request. The link latencies are almost negligible and the Paxos part of the request servicing should only come out to a few milliseconds. The service time again can be splits into 2 dominant factors:

Code execution time of the state machine itself

Database access times

* + - * Results:



Client end to end latency



1. Conclusion